Coda: A Highly Available File System for a Distributed Workstation Environment

M. Satyanarayanan
James J. Kistler
Puneet Kumar
Maria E. Okasaki
Ellen H. Siegel
David C. Steere

School of Computer Science
Carnegie Mellon University
17 November 1989

Abstract

Coda is a file system for a large-scale distributed computing environment composed of Unix workstations. It provides resiliency to server and network failures through the use of two distinct but complementary mechanisms. One mechanism, *server replication,* stores copies of a file at multiple servers. The other mechanism, *disconnected operation,* is a mode of execution in which a caching site temporarily assumes the role of a replication site. Disconnected operation is particularly useful for supporting portable workstations. The design of Coda optimizes for availability and performance, and strives to provide the highest degree of consistency attainable in the light of these objectives. Measurements from a prototype show that the performance cost of providing high availability in Coda is reasonable.

This research was supported by the National Science Foundation (Contract No. CCR-8657907), Defense Advanced Research Projects Agency (Order No. 4976, Contract F33615-87-C-1499), the IBM Corporation (Faculty Development Award and Graduate Fellowship), and Digital Equipment Corporation (Equipment Grant). The views and conclusions in this document are those of the authors and do not represent the official policies of the funding agencies or Carnegie Mellon University.

1. Introduction

A location-transparent distributed file system based on the Unix[1] file system model is a valuable mechanism for collaboration between physically dispersed users. This is particularly true in a distributed workstation environment where the primary activities are education, research and software development. The *Andrew File System* (AFS) is a highly successful realization of such a mechanism for a campus-sized user community [9, 15]. Positive experience with AFS has motivated the recent work on extending it nationwide [17]. The importance of a shared Unix file system for a distributed workstation environment is further confirmed by many other efforts in industry and academia [16].

The work described in this paper arose from our extensive experience as implementors and users of AFS over the past five years. On the one hand, we were pleased with the functionality, performance, and ease of administration of AFS. At the same time we were concerned with its vulnerability to failures of servers and network components. Such failures can seriously inconvenience many users for time periods ranging from a few minutes to many hours. At least a few such outages occur somewhere in our system every day.

The *Coda File System* is a descendant of AFS that is substantially more resilient to failures. The ideal that Coda strives for is *constant data availability,* allowing a user to continue working regardless of failures elsewhere in the system. Our goal is to provide users with the benefits of a shared data repository, but to allow them to rely entirely on local resources when that repository is partially or totally inaccessible.

A related goal of Coda is to gracefully integrate the use of AFS with portable computers. At present, users manually copy relevant files from AFS, use the machine while isolated from the network, and manually copy updated files back to AFS upon reconnection. These users are effectively performing manual caching of files with write-back on reconnection. If one views the disconnection from AFS as a deliberately-induced failure, it is clear that a mechanism for supporting portable machines in isolation is also a mechanism for fault tolerance.

Although this paper focuses on Coda, the problem of data availability is not specific to it. Growth in scale and complexity inevitably results in more components that can fail, and more hardware and software interactions that can manifest themselves as failures. Consequently, the large-scale distributed file systems of the future will have to provide continued access to data in the face of temporary failures if they are to remain viable.

In the rest of this paper we describe the design and implementation of Coda. We motivate our design choices in Section 2, present the design in Sections 3 through 5, and elaborate on the key data structures and protocols in Section 6. We describe the status and performance of a Coda prototype in Sections 7 and 8. We conclude with an examination of related research in Section 9 and a summary of the paper in Section 10.

2. Design Rationale

Three fundamental factors influenced the design of Coda. These are

- Our desire to produce a highly scalable system.
- The range of failures we wished to address.
- The need to emulate Unix file semantics.

In the course of our design, we discovered that the constraints imposed by these factors often conflicted with each other. The current design of Coda is therefore a compromise that, in our judgment, best suits our usage environment.

[1]Unix is a trademark of AT&T.

2.1. Scalability

A *scalable* distributed system is one that can easily cope with the addition of users and sites. Growth has economic, performance and administrative consequences. Our goal was to build a system whose growth would incur as little expense, performance degradation, and administrative complexity as possible. Since this goal was also the major focus of Coda's ancestor, AFS, we tried to preserve much of its design.

In AFS, a small set of trusted servers jointly provide a storage repository shared by a much larger number of untrusted clients. To maximize client-server ratio, most load is borne by clients. Only functions essential to integrity or security are performed by servers. *Caching* is the key to scalability in AFS. The operating system on each client intercepts open and close file system calls[2] and forwards them to a cache-management process called *Venus.* After a file is opened, read and write operations on it bypass Venus. Venus contacts a server only on a cache miss on open, or on a close after modification. In both cases the file is transferred in its entirety. Cache coherence is maintained by a *callback* mechanism, whereby servers notify workstations of changes to cached files. Clients dynamically determine the location of files on servers and cache this information.

The highly dynamic nature of AFS enhances its scalability. There are very few static bindings that require atomic, system-wide updates for AFS to function correctly. A workstation with a small disk can potentially access any file in AFS by name. One can move to any other workstation and effortlessly access one's files from there. Adding a new workstation merely involves connecting it to the network and assigning it an address. Workstations can be turned off or physically relocated at any time without fear of inconveniencing other users. Only a small operational staff is required to monitor and service the relatively few AFS servers. Backup is needed only on the servers, since workstation disks are merely used as caches. Files can be easily moved between servers during normal operation without inconveniencing users.

Coda retains many of the features of AFS that contribute to its scalability and security

- It uses the model of a few trusted servers and many untrusted clients.
- Clients cache entire files on their local disks. From the perspective of Coda, whole-file transfer also offers a degree of intrinsic resiliency. Once a file is cached and open at a client, it is immune to server and network failures. Caching on local disks is also consistent with our goal of supporting portable machines.
- Cache coherence is maintained by the use of callbacks. But, as described later in the paper, the maintenance of callbacks is more complex in Coda than in AFS.
- Clients dynamically map files to servers and cache this information.
- It uses token-based authentication and end-to-end encryption integrated with its communication mechanism [13].

2.2. Range of Failures

To achieve our goal of continued client operation in the face of failures, we had two strategies available to us. The first was to use replication across servers to render the shared storage repository more reliable. The second was to make each client capable of fully autonomous operation if the repository failed. Each of these strategies improves availability, but neither is adequate alone.

Enhancing the availability of the shared storage repository increases the availability of all shared data. It protects against individual server failures and some network failures. Unfortunately it does not help if all servers fail, or if all of them are inaccessible due to a total partition of the client. A special case of the latter is the use of a portable computer when detached from the network.

Making each client fully autonomous is infeasible. The disk storage capacity of a client is a small fraction of the

[2]Directories are also cached on clients, but modifications to them are immediately propagated to servers. For ease of exposition we confine our discussion to files in this section.

total shared data. This strategy is also inconsistent with our model of treating each client's disk merely as a cache. It represents a return to the model of isolated personal computers rather than a collection of workstations sharing a file system. The advantages of mobility, and the ability of any user to use any workstation as his own, are lost. Yet temporary autonomy seems acceptable for brief periods of time, of the order of minutes or hours, while a user is active at a client.

In the light of these considerations we decided to use a combination of the two strategies to cover a broad range of failures. Coda uses *server replication,* or the storing of copies of files at multiple servers, to provide a shared storage repository of higher availability than AFS. A client relies on server replication as long as it remains in contact with at least one server. When no server can be contacted the client resorts to *disconnected operation,* a mode of execution in which the client relies solely on cached data. We regard involuntary disconnected operation as a measure of last resort and revert to normal operation at the earliest opportunity. A portable client that is isolated from the network is effectively operating disconnected.

Our need to handle network failures meant that we had to address the difficult issue of consistency guarantees across partitions. In the terminology of Davidson et al [3], we had to decide whether to use a *pessimistic* replication strategy, providing strict consistency, or an *optimistic* strategy, providing higher availability. The former class of strategies avoid update conflicts by restricting modifications to at most one partition. The latter allows updates in every partition, but detects and resolves conflicting updates after they occur.

We chose to use an optimistic strategy for three reasons. First, and most important, such an approach provides higher availability. Second, we saw no clean way of supporting portable workstations using a pessimistic strategy. Third, it is widely believed that write sharing between users is relatively infrequent in academic Unix environments. Consequently, conflicting updates are likely to be rare. We guarantee detection and confinement of these conflicts, and try to do this as soon after their occurence as possible.

The specific replication strategy we chose is an adaptation of that originally proposed by Locus [19]. *Coda version vectors* (CVVs), similar in concept but distinct in detail from version vectors described by Parker et al [10], are used to detect *write-write* conflicts on individual files. We did not choose the more powerful optimistic strategy proposed by Davidson [2], even though it is capable of detecting *read-write* conflicts across multiple files. We were concerned with the complexity of the latter strategy and questioned its value in a Unix environment where multi-file transactional guarantees are absent. We also felt that inconsistencies due to conflicting updates should be brought to the attention of users rather than being rolled back by the system.

To summarize, Coda enhances availability both by the replication of files across servers, as well as by the ability of clients to operate entirely out of their caches. Both mechanisms depend upon an optimistic strategy for detection of update conflicts in the presence of network partitions. Although these mechanisms are complementary, they can be used independently of each other. For example, a Coda installation might choose to exploit the benefits of disconnected operation without incurring the CPU and disk storage overhead of server replication.

2.3. Unix Emulation

Our ideal is to make Coda appear to be a giant, failure-proof shared Unix file system. Unfortunately, realizing this ideal requires strict adherence to *one-copy Unix semantics.* This implies that every modification to every byte of a file has to be immediately and permanently visible to every client. Such a requirement is obviously in conflict with our goals of scalability and availability. We have therefore relaxed the accuracy with which we emulate Unix semantics, and have settled for an approximation that can be implemented in a scalable and available manner. We have drawn upon two lessons from AFS to develop this approximation, and believe that it will satisfy the vast majority of users and applications in our environment.

2.3.1. AFS-1 Semantics

The first lesson was that propagating changes at the granularity of file opens and closes was adequate for virtually all applications in our environment. The initial prototype of AFS (AFS-1) revalidated cached files on each open, and propagated modified files when they were closed. A successful open implied that the resulting copy of the file was the latest in the system.

We can precisely state the currency guarantees offered by this model by considering a client C operating on a file F whose custodian is server S. Let $latest(F, S)$ denote the fact that the current value of F at C is the same as that at S. Let $failure(S)$ denote failure of the current operation by C on S, and $updated(F, S)$ denote a successful propagation of C's copy of F to S. Then the currency guarantees provided by open and close operations at C can be expressed as follows:

open success	$latest(F, S)$
open failure	$failure(S)$
close success	$updated(F, S)$
close failure	$failure(S)$

2.3.2. AFS-2 Semantics

The second lesson we learned was that the slightly weaker currency guarantees provided by the callback mechanism of the revised AFS design (AFS-2) were acceptable. A callback is established as a side effect of file fetch or cache validation. It is a guarantee by a server that it will notify the client on the first modification of the file by any other client. If this notification happens, the server is said to have *broken* the callback[3]. Once broken, the callback has to be reestablished by the client. But, as a result of a network failure, a server's attempt to break a callback may not be noticed by a client. We refer to such an event as a *lost* callback. Because of a lost callback, a client may continue to use a cached copy of a file for up to τ seconds after the file was updated elsewhere. τ is a parameter of the system, and is typically of the order of a few minutes.

We can characterize the currency guarantees of AFS with callbacks by extending our notation. $latest(F, S, t)$ now denotes the fact that the current value of F at C is the same as that at S at some instant in the last t seconds. In particular, $latest(F, S, 0)$ means F is currently identical at C and S. If we indicate the loss of a callback from S to C during the last t seconds by $lostcallback(S, t)$, and the presence of F in C's cache prior to the current operation by $incache(F)$, the resulting currency guarantees can be stated thus:

open success	$latest(F, S, 0) \vee (latest(F, S, \tau) \wedge lostcallback(S, \tau) \wedge incache(F))$
all others	as for AFS-1

2.3.3. Coda Semantics

In Coda, the single server S is replaced by a set of servers $\overline{S}$. C maintains the subset $\overline{s}$ of $\overline{S}$ that it was able to contact on the most recent remote operation. $\overline{s}$ is reevaluated at least once every τ seconds. When $\overline{s}$ is empty, C is operating disconnected. The intuitive currency guarantee offered by a successful Coda open is that it yields the most recent copy of F among the set of currently accessible servers. If no server is accessible, the cached copy of F is used. A successful close indicates that the file has been propagated to the set of currently accessible servers, or that no server is available and the file has been marked for propagation at the earliest opportunity.

The use of callbacks and an optimistic replication scheme weakens these intuitive currency guarantees. A more precise statement of the guarantees can be made by further extension of our notation. $latest(F, \overline{s}, t)$ now denotes the fact that the current value of F at C was the latest across all servers in $\overline{s}$ at some instant in the last t seconds. It also

[3] Unfortunately the terminology is a little confusing. As used in the AFS literature, ''callback'' is a noun rather than a verb, and is an abbreviation for ''callback promise''.

denotes the fact that there were no conflicts among the copies of $\bar{s}$ at that instant. *lostcallback*$(\bar{s}, t)$ now means that a callback from some member of $\bar{s}$ to C was lost in the last t seconds. *updated*$(F, \bar{s})$ means that the current value F at C was successfully propagated to all members of $\bar{s}$. *conflict*$(F, \bar{s})$ means that the values of F at $\bar{s}$ are currently in conflict. Using this notation, the currency guarantees offered by Coda operations can be expressed as follows:

open success	$(\bar{s} \neq \emptyset \wedge (latest(F, \bar{s}, 0) \vee (latest(F, \bar{s}, \tau) \wedge lostcallback(\bar{s}, \tau) \wedge incache(F))))$
	$\vee\ (\bar{s} = \emptyset \wedge incache(F))$
open failure	$(\bar{s} \neq \emptyset \wedge conflict(F, \bar{s}))$
	$\vee\ (\bar{s} = \emptyset \wedge \neg\, incache(F))$
close success	$(\bar{s} \neq \emptyset \wedge updated(F, \bar{s}))$
	$\vee\ (\bar{s} = \emptyset)$
close failure	$(\bar{s} \neq \emptyset \wedge conflict(F, \bar{s}))$

Although we believe that the currency guarantees of Coda are adequate for a typical academic or research environment, they may be too weak for some applications. Databases are a class of applications that we specifically do not attempt to support in Coda. Our view is that a database for a large-scale distributed environment should be implemented as a separate system rather than being built on top of a distributed file system.

3. Server Replication

The unit of replication in Coda is a *volume*, a set of files and directories located on one server and forming a partial subtree of the shared name space[4]. Each file and directory in Coda has a unique low-level *file identifier* (FID), a component of which identifies the parent volume. All replicas of an object have the same FID.

The set of servers with replicas of a volume constitute its *volume storage group* (VSG). The degree of replication and the identity of the replication sites are specified when a volume is created and are stored in a *volume replication database* that is present at every server. Although these parameters can be changed later, we do not anticipate such changes to be frequent. For every volume from which it has cached data, Venus (the client cache manager) keeps track of the subset of the VSG that is currently accessible. This subset is called the *accessible volume storage group* (AVSG). Different clients may have different AVSGs for the same volume at a given instant. In the notation of Section 2.3.3, the VSG and AVSG correspond to $\bar{S}$ and $\bar{s}$ respectively.

3.1. Strategy

The replication strategy we use is a variant of the *read-one, write-all* approach. When servicing a cache miss, a client obtains data from one member of its AVSG called the *preferred server.* The preferred server can be chosen at random or on the basis of performance criteria such as physical proximity, server load, or server CPU power. Although data is transferred only from one server, the other servers are contacted by the client to verify that the preferred server does indeed have the latest copy of the data. If this is not the case, the member of the AVSG with the latest copy is made the preferred site, the data is refetched, and the AVSG is notified that some of its members have stale replicas. As a side effect, a callback is established with the preferred server.

When a file is closed after modification it is transferred in parallel to all members of the AVSG. This approach is simple to implement and maximizes the probability that every replication site has current data at all times. Server CPU load is minimized because the burden of data propagation is on the client rather than the server. This in turn improves scalability, since the server CPU is the bottleneck in many distributed file systems.

---

[4]Coda also supports non-replicated volumes and volumes with read-only replicas, a feature inherited from AFS. We restrict our discussion here to volumes with read-write replicas.

Since our replication scheme is optimistic, we have to check for conflicts on each server operation. We also require that server modifications be made in a manner that will enable future conflicts to be detected. These issues are further discussed in Section 6, which describes the data structures and protocols used in server replication.

At present, a server performs no explicit remote actions upon recovery from a crash. Rather, it depends upon clients to notify it of stale or conflicting data. Although this lazy strategy does not violate our currency guarantees, it does increase the chances of a future conflict. A better approach, which we plan to adopt in the future, is for a recovering server to contact other servers to bring itself up to date.

3.2. Cache Coherence

The Coda currency guarantees stated in Section 2.3 require that a client recognize three kinds of events no later than τ seconds after their occurence:

- Enlargement of an AVSG (implying accessibility of a previously inaccessible server).
- Shrinking of an AVSG (implying inaccessibility of a previously accessible server).
- A lost callback event.

Venus detects enlargement of an AVSG by trying to contact missing members of the VSG once every τ seconds. If an AVSG enlarges, cached objects from the volume may no longer be the latest copy in the new AVSG. Hence the client drops callbacks on these objects. The next reference to any of these objects will cause the enlarged AVSG to be contacted and a newer copy to be fetched (if one exists).

Venus detects shrinking of an AVSG by probing its members once every τ seconds. Shrinking is detected earlier if a normal operation on the AVSG fails. If the shrinking is caused by loss of the preferred server, Venus drops its callbacks from it. Otherwise they remain valid. It is important to note that Venus only probes those servers from which it has cached data; it does not probe other servers, nor does it ever probe other clients. This fact, combined with the relative infrequency of probes (τ being ten minutes in our current implementation), ensures that probes are not an obstacle to the scalability of the system.

If Venus were to place callbacks at all members of its AVSG, the probe to detect AVSG shrinking would also detect lost callback events. Since maintaining callback state at all servers is expensive, Venus only maintains a callback at the preferred server. The probe to the preferred server detects lost callback events from it.

But maintaining callbacks only at one server introduces a new problem. The preferred server for one client need not necessarily be in the AVSG of another client[5]. Hence an update of an object by the second client may not cause a callback for the object at the first client to be broken.

To detect updates missed by its preferred server, each probe by Venus requests the *volume version vector (volume CVV)* for every volume from which it has cached data. A volume CVV is similar to a file or directory CVV, but summarizes update information on the entire volume. It is updated as a side-effect of every operation that modifies the volume. A mismatch in the volume CVVs indicates that some AVSG members missed an update. Although the missed update may not have been to an object in the cache, Venus conservatively drops its callbacks on all objects from the volume.

---

[5]This can happen, for example, due to non-transitivity of network communication.

3.3. Parallel Communication

Because of server replication, each remote operation in Coda typically requires multiple sites to be contacted. If this were done serially, latency would be degraded intolerably. Venus therefore communicates with replication sites in parallel, using the *MultiRPC* parallel remote procedure call mechanism [14]. The original version of MultiRPC provided logical parallelism but did not use multicast capability at the media level. Since we were particularly concerned about the latency and network load caused by shipping large files to multiple sites, we have extended MultiRPC to use hardware multicast. But we view multicast as an optimization rather than a fundamental requirement, and Coda retains the ability to use non-multicast MultiRPC.

4. Disconnected Operation

Disconnected operation begins at a client when no member of a VSG is accessible. Clients view it as a temporary state and revert to normal operation at the earliest opportunity. A client may be operating disconnected with respect to some volumes, but not others. Disconnected operation is transparent to a user unless a cache miss occurs. Return to normal operation is also transparent, unless a conflict is detected.

4.1. Cache Misses

In normal operation a cache miss is transparent to the user, and only imposes a performance penalty. But in disconnnected operation a miss impedes computation until normal operation is resumed or until the user aborts the corresponding file system call. Consequently it is important to avoid cache misses during disconnected operation.

During brief failures, the normal LRU caching policy of Venus may be adequate to avoid cache misses to a disconnected volume. This is most likely to be true if a user is editing or programming and has been engaged in this activity long enough to fill his cache with relevant files. But it is unlikely that a client could operate disconnected for an extended period of time without generating references to files that are not in the cache.

Coda therefore allows a user to specify a prioritized list of files and directories that Venus should strive to retain in the cache. Objects of the highest priority level are *sticky* and must be retained at all times. As long as the local disk is large enough to accomodate all sticky files and directories, the user is assured that he can always access them. Since it is often difficult to know exactly what file references are generated by a certain set of high-level user actions, Coda provides the ability for a user to bracket a sequence of high-level actions and for Venus to note the file references generated during these actions.

4.2. Reintegration

When disconnected operation ends, a process of *reintegration* begins. For each cached file or directory that has been created, deleted, or modified during disconnected operation Venus executes a sequence of update operations to make AVSG replicas identical to the cached copy. Reintegration proceeds top-down, from the root of each cached volume to its leaves.

Update operations during reintegration may fail for one of two reasons. First, there may be no authentication tokens which Venus can use to securely communicate with AVSG members. Second, inconsistencies may be detected due to updates to AVSG replicas. Given our model of servers rather than clients being dependable storage repositories, we felt that the proper approach to handling these situations was to find a temporary home on servers for the data in question and to rely on a user to resolve the problem later.

The temporary repository is realized as a *covolume* for every replica of every volume in Coda. Covolumes are similar in spirit to *lost+found* directories in Unix. Having a covolume per replica allows us to reintegrate as soon as any VSG site becomes available. The storage overhead of this approach is usually small, since a covolume is almost always empty.

4.3. Voluntary Disconnection

Disconnected operation can also occur *voluntarily*, when a client is deliberately disconnected from the network. This might happen, for instance, when a user takes a portable machine with him on his travels. With a large disk cache the user can operate isolated from Coda servers for an extended period of time. The file name space he sees is unchanged, but he has to be careful to restrict his references to cached files and directories. From time to time, he may reconnect his client to the network, thereby propagating his modifications to Coda servers.

By providing the ability to move seamlessly between zones of normal and disconnected operation, Coda may be able to simplify the use of cordless network technologies such as cellular telephone, packet radio, or infra-red communication in distributed file systems. Although such technologies provide client mobility, they often have intrinsic limitations such as short range, inability to operate inside buildings with steel frames, or line-of-sight constraints. These shortcomings are reduced in significance if clients are capable of autonomous operation.

5. Conflict Resolution

When a conflict is detected, Coda first attempts to resolve it automatically. Since Unix files are untyped byte streams there is, in general, no information to automate their resolution. Directories, on the other hand, are objects whose semantics are completely known. Consequently, their resolution can sometimes be automated. If automated resolution is not possible, Coda marks all accessible replicas of the object inconsistent. This ensures damage containment since normal operations on these replicas will fail. A user will have to manually resolve the problem using a repair tool.

5.1. Automated Resolution

The semantics of a Coda directory is that it is a list of <*name, FID*> pairs with two modification operations, create and delete, that can act on the list. Status modifications, such as protection changes, can also be made on the directory. The resolution procedure for Coda directories is similar to that of Locus [6, 19]. There are three classes of conflicts involving directories that are not amenable to automated resolution. One class, *update/update* conflict, is exemplified by protection modifications to partitioned replicas of a directory. The second class, *remove/update* conflict, involves updating an object in one partitition and removing it in another. The third class, *name/name* conflict, arises when new objects with identical names are created in partitioned replicas of a directory. All other directory conflicts can be automatically resolved by a compensating sequence of create or delete operations.

5.2. Repair Tool

The Coda repair tool allows users to manually resolve conflicts. It uses a special interface to Venus so that file requests from the tool are distinguishable from normal file requests. This enables the tool to overwrite inconsistent files and to perform directory operations on inconsistent directories, subject to normal access restrictions. To assist the user, each replica of an inconsistent object is made available in read-only form. Since these read-only copies are not themselves inconsistent, normal Unix applications such as editors may be used to examine them.

6. Replica Management

We now examine replica management at the next level of detail, focusing on the data structures and protocols used in server replication. We begin with an abstract characterization of replica states in Section 6.1, and then describe an approximation that can be efficiently realized in Section 6.2. This approximation is conservative, in that it may occasionally indicate a conflict where none exists but will never fail to detect a genuine conflict. Finally we describe the protocols that modify replicas in Section 6.3.

6.1. State Characterization

Each modification on a server can be conceptually tagged with a unique *storeid* by the client performing the operation. If a server were to maintain a chronological sequence of the storeids of an object, it would possess the entire *update history* of the object at that server.

The *latest storeid (LSID)* in the update history of a replica can be used to characterize its state relative to another replica. Two replicas, A and B, are said to be *equal* if their LSIDs are identical. Equality represents a situation where the most recent update to both replicas was the same. If B's LSID is distinct from A's LSID but is present in A's history, the replica at A is said to *dominate* the replica at B. This situation may also be described by saying B is *submissive* to A. In this situation, both sites have received a common update at some point in the past, but the submissive site has received no updates thereafter. The replicas are said to be *inconsistent* if neither A's LSID nor B's LSID is present in the other's update history. Inconsistency represents a situation where updates were made to a replica by a client that was ignorant of updates made to another replica.

In the case of files, a submissive replica directly corresponds to our intuitive notion of stale data. Hence Coda always provides access to the dominant replica of a file among a set of accessible replicas. An inconsistency among file replicas arises from genuine update conflicts. In such a situation, Coda immediately marks all accessible replicas in a manner that causes normal operations on them to fail.

The situation is more complex in the case of directories, because the update history of a directory does not capture activity in its children. Consequently, update histories can only be used conservatively in characterizing the states of directory replicas. Replicas whose update histories are equal are indeed identical, but replicas with unequal update histories are potentially in conflict.

6.2. State Representation

Since it would be impractical to maintain the entire update history of a replica, Coda maintains an approximation to it. The approximation consists of the current length of the update history and its LSID. The LSID is composed of an identifier unique to each client, concatenated with a monotonically increasing integer[6]. A replication site also maintains an estimate of the length of the update history of every other replica. A vector containing these length estimates constitutes the CVV at this site. An estimate is always conservative. In other words, a site may fail to notice an update made to a replica, but it will never erroneously assume that the replica was updated. A site's estimate of updates to itself will be accurate as long it has the ability to make local modifications in a manner that is atomic and permanent.

Coda compares the states of replicas using their LSIDs and CVVs. When two replicas, A and B, are compared the outcome is constrained to be one of four possibilities:

- *strong equality*, where LSID_A is identical to LSID_B, and CVV_A is identical to CVV_B.
- *weak equality*, where LSID_A is identical to LSID_B, but CVV_A and CVV_B are not identical.
- *dominance/submission*, where LSID_A is different from LSID_B, and every element of CVV_A is greater than or equal to the corresponding element of CVV_B (or vice versa).
- *inconsistency*, where LSID_A is different from LSID_B, and some elements of CVV_A are greater than, but other elements are less than, the corresponding elements of CVV_B.

Strong equality corresponds to a situation where a client successfully updates A and B, and each replica is certain of the other's update. Weak equality arises when the update succeeds at both sites, but this fact is not known to both replicas. Together, strong and weak equality correspond to the notion of replica equality defined in terms of update histories in Section 6.1. The pairwise comparisons defined here can be easily generalized to set comparisons.

---

[6]In our implementation these entities are the IP-address of a workstation and a logical timestamp.

6.3. State Transformation

There are four classes of operations in Coda that can change the state of a server replica:

- *update* extends the update history by a new, hitherto unused, storeid.
- *force* logically replays those updates made to a dominant site that are missing from a submissive one.
- *repair* resembles update, but is used to return a set of replicas previously marked inconsistent to normal use.
- *migrate* saves copies of objects involved in unsuccessful updates resulting from disconnected operation for future repair.

We describe the details of these classes of operations in the following sections. When we refer to file or directory status in these sections, we include the CVV and LSID.

6.3.1. Update

Update is, by far, the most frequent class of mutating operations. Every common client-server interaction that involves modification of data or status at the server falls into this class. Examples include file store, file and directory creation and deletion, protection change, and link creation. In updates to existing objects the protocol consists of two phases, with the client acting as initiator and coordinator. In the first phase, each AVSG site checks the LSID and CVV presented by the client. If the check succeeds, the site performs the requested semantic action such as the transfer of data in the case of a file store. In the second phase, each AVSG site records the client's view of which sites executed the previous phase successfully. In updates where a new object has to be created, these two phases are preceded by a phase where a new FID is allocated by the preferred server.

The check at an AVSG site in the first phase succeeds for files if the cached and server copies are equal or if the cached copy dominates. Cached-copy dominance is acceptable for files since an update for a submissive site is logically equivalent to a force that brings its replica into equality followed by the actual update. Since new file data merely overwrites existing data, we omit the force. For directories, the check succeeds only when the two copies are equal. An unsuccessful check of either type of object by any AVSG site causes the client to pause the operation and invoke the resolution subsystem at the AVSG. If the resolution subsystem is able to automatically fix the problem, the client restarts the paused operation. Otherwise the operation is aborted and an error is returned to the user. A successful check causes the server to atomically perform the semantic action, and to commit a new LSID (sent by the client in the first phase) and a tentative CVV that is identical to the client's except for one additional update at this server.

The client examines the replies from the first phase and distributes a final CVV. The latter is identical to the CVV of the first phase except that it indicates one additional update at each responding server. Servers that receive this information replace their tentative CVVs by the final CVV. At an AVSG site that crashed or was partitioned between the first and second phases, the tentative CVV remains unchanged.

Since update is frequent, it is important to optimize its performance. The total number of messages and latency are reduced by communicating with AVSG members in parallel. Latency is further reduced by having Venus return control to the user at the end of the first phase. Server throughput is increased by the use of batching and piggybacking in the second phase.

6.3.2. Force

A force operation is a server-to-server interaction, with a client playing no part except to set in motion a sequence of events that leads to the force. For example, a force operation may occur as a result of Venus notifying its AVSG that it has detected an inequality during a file fetch. It may also occur when the system determines that a directory conflict can be resolved by a sequence of forces. Force operations may also arise on server crash recovery, when a server brings itself up to date.

A force of a file merely consists of atomically copying its data and status from the dominant to the submissive site.

But a force of a directory is more complex. The ideal operation would be one that rendered the subtrees rooted at the directory replicas identical. The subtrees would be exclusively locked for the entire duration of the force, and all changes would be atomic. Unfortunately this is impractical, especially if the subtrees in question are deep. Consequently our approach is to lock, and atomically apply changes, a directory at a time.

This approach does not violate our ability to detect genuine conflicts for two reasons. First, directories only contain information about immediate descendants. Second, when creating an entry for a new object, we first make it point to a *runt* replica which has a CVV that will always be submissive. A failure to the forcing server could occur after the creation, but before the force, of the runt. But any subsequent attempt to access the runt would result in detection of inequality.

6.3.3. Repair and Migrate

Both repair and migrate are relatively rare operations. A repair operation is used to fix inconsistency and proceeds in two phases, similar to an update. A migrate operation is used to place an object in conflict at the end of disconnected operation in a covolume on a server. The server replica is marked inconsistent, and accesses to the object will fail until it is repaired.

7. Implementation Status

Our goal in implementing Coda is to explore its overall feasibility and to obtain feedback on its design. The prototype implementation runs on IBM RTs, and is functional in most respects. One can sit down at a Coda client and execute Unix applications without recompilation or relinking. Execution continues transparently when contact is lost with a server due to a crash or network failure. In the absence of failures, using a Coda client feels no different from using an AFS client. The primary areas where our implementation is incomplete are conflict resolution, the implemention of sticky files, and certain aspects of reintegration. We expect to complete these shortly.

We use the Camelot transaction facility [18] to obtain atomicity and permanence of server operations. Our use of Camelot is restricted to single-site, top-level transactions. We do not use nested or distributed transactions in our implementation. To reduce the latency caused by synchronous writes to the Camelot log, we have built a battery-backed ramdisk for each of our servers.

To test the resiliency of the system we have built an emulator that induces controlled and repeatable failures in Coda. It consists of an interactive front-end that runs on a single machine, and emulation routines that are invoked by the communication package at every Venus and file server.

8. Performance Evaluation

In this section we present measurements that reflect on the design and implementation of the Coda prototype. Our discussion focuses on four questions:

- What is the effect of server replication?
- How does Coda behave under load?
- How important is multicast?
- How useful is a ramdisk for logging?

We have spent little effort until now on tuning the low level aspects of our implementation. It is likely that a refined implementation will show noticeable performance improvement. This should be kept in mind in interpreting the results reported here.

8.1. Methodology and Configuration

Our evaluation is based on the Andrew benchmark [7], which operates on a collection of files constituting the source code of a Unix application. An instance of the benchmark generates as much network and server load as 5 typical AFS users. We use the term *load* to refer to the number of clients simultaneously running this benchmark.

The input to the benchmark is a subtree of 70 files totalling 200 Kbytes in size. There are five distinct phases in the benchmark: *MakeDir,* which constructs a target subtree that is identical in structure to the source subtree; *Copy,* which copies every file from the source subtree to the target subtree; *ScanDir,* which recursively traverses the target subtree and examines the status of every file in it; *ReadAll,* which scans every byte of every file in the target subtree twice; and *Make,* which compiles and links all the files in the target subtree. The ScanDir and ReadAll phases reap the most benefit from caching, and hence show the least variation in our experiments.

The clients and servers used in our experiments were IBM RT/APCs with 12MB of main memory and 70MB disks, running the Mach operating system, and communicating on an Ethernet with no intervening routers. Each server had an additional 400MB disk on which Coda volumes were stored. Each experiment was repeated at least 3 times, with careful experimental control. In no case was the variance in any measured quantity more than a few percent of the corresponding mean.

Our reference point is Coda replicated at three servers with Ethernet multicast enabled, a ramdisk at each server for the Camelot log, and a warm cache[7]. This configuration is labelled ‘‘Coda:3’’ in the graphs. For comparison, we also ran experiments on the same hardware and operating system with Coda replicated at two and one servers (‘‘Coda:2’’ and ‘‘Coda:1’’ respectively), with Coda non-replicated (‘‘Coda:NoRep’’), with the current release of AFS (‘‘AFS’’), and with the local Unix file system of a client (‘‘Unix’’).

8.2. Effect of Server Replication

In the absence of failures, we would like Coda’s performance to be minimally affected by its high availability mechanisms. Server replication is the primary source of performance degradation, since it involves a more complex protocol as well as data transfer to multiple sites. Camelot is another potential source of performance degradation.

Figure 1 shows the effect of server replication. Without replication, Coda takes 21% longer than the local Unix file system to run the benchmark. This is essentially the same as that of the current production version of AFS. With replication at one, two and three servers Coda takes 22%, 26% and 27% longer than Unix.

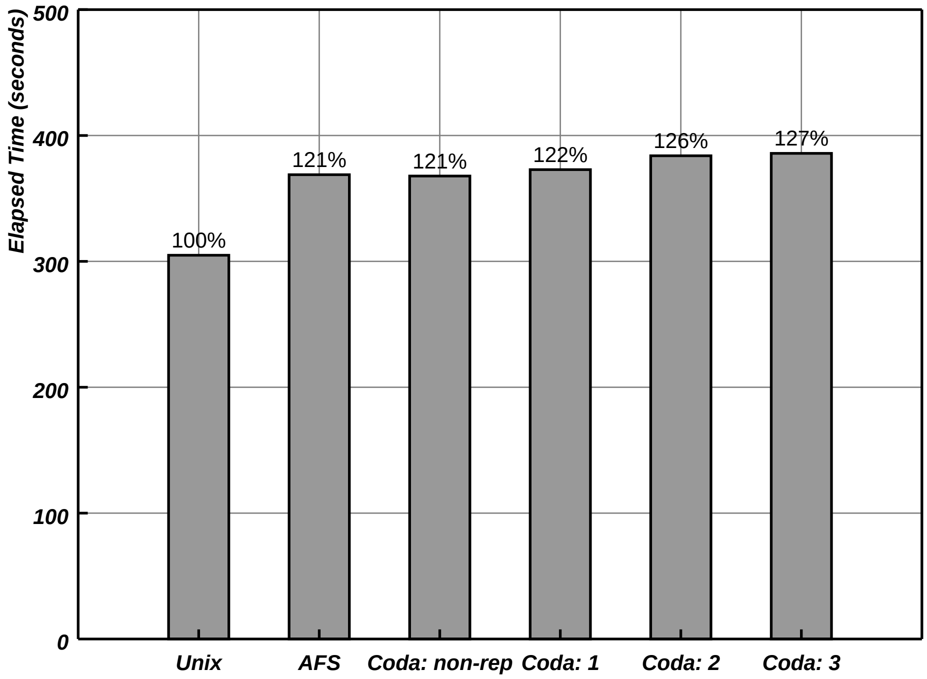
As Table 1 shows, the Copy phase of the benchmark is most affected by replication since it benefits least from caching. On a non-replicated Coda volume this phase takes 73% longer than on Unix. On a volume replicated at one, two and three servers it takes 91%, 109%, and 118% longer. For comparison, AFS takes 82% longer. Table 1 also shows that the ScanDir phase is noticeably longer in Coda than in AFS. This is because the Coda cache manager is a user process, while the AFS cache manager is inside the kernel. Consequently, Coda incurs additional overhead in translating a pathname, even if valid cached copies of all components of the pathname are cached.

8.3. Behavior under Load

How does Coda perform when multiple workstations use it simultaneously? Figure 2 and Table 2 show the total elapsed time of the benchmark as a function of load. As load is increased from 1 to 10, the time for the benchmark increases from 100% to 170%. As mentioned earlier, one load unit roughly corresponds to five typical AFS users. In contrast, the benchmark time for AFS only increases from 100% to 116% as load is increased from 1 to 10.

Server CPU utilization is the primary contributor to the difference in behavior between Coda and AFS under load.

[7]Our measurments show that the main effect of a cold cache is to lengthen the time of the Copy phase by 23% at a load of one.



This graph shows the total elapsed time of the Andrew benchmark at a load of one as a function of varying the degree of server replication. It also compares Coda with AFS and Unix. This data is presented in more detail in Table 1.

Figure 1: Effect of Replication on Elapsed Time

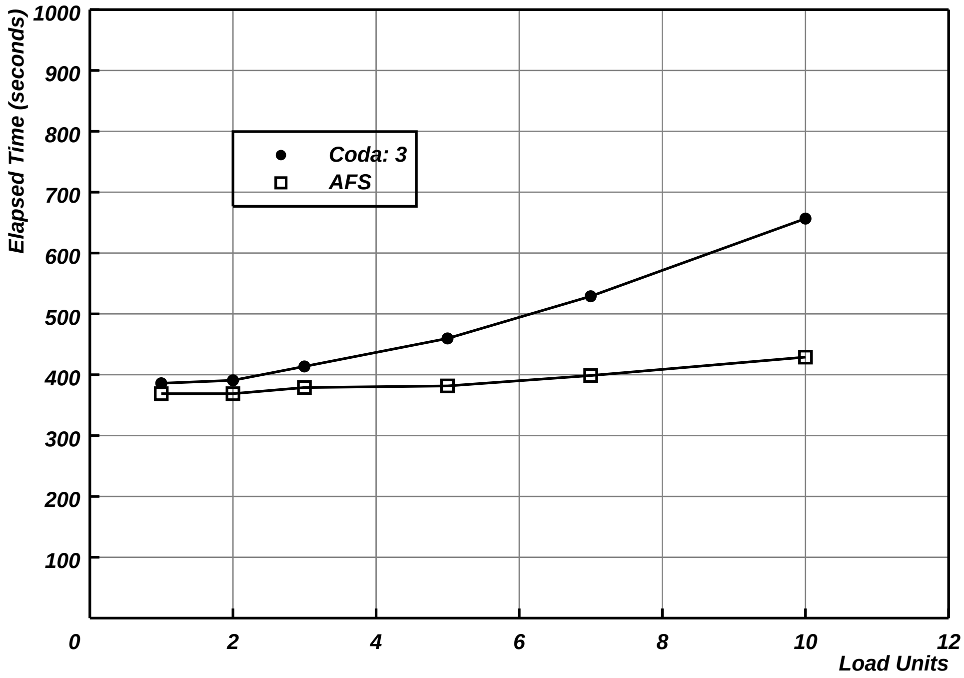
Configuration	Load Units	MakeDir	Copy	ScanDir	ReadAll	Make	Total
Coda: 3	1	5 (1)	48 (1)	33 (1)	52 (1)	248 (5)	386 (5)
Coda: 2	1	5 (1)	46 (2)	32 (1)	52 (1)	247 (1)	384 (3)
Coda: 1	1	5 (0)	42 (1)	32 (1)	52 (1)	242 (2)	373 (3)
Coda: NoRep	1	4 (0)	38 (1)	32 (0)	52 (1)	241 (1)	368 (2)
AFS	1	7 (3)	40 (1)	27 (2)	53 (0)	242 (4)	369 (7)
Unix	1	5 (2)	22 (1)	21 (1)	36 (1)	221 (1)	305 (1)

This table presents the elapsed time of the phases of the Andrew benchmark for a variety of configurations. Each time reported is in seconds, and is the mean of 3 trials. Numbers in parentheses are standard deviations.

Table 1: Effect of Replication on Elapsed Time

Three factors contribute to increased server CPU utilization in Coda. The first factor is, of course, the overhead due to replication. The second is our use of Camelot. The third is the lack of tuning of the Coda implementation.

Figure 3 shows the relative contributions of each of these factors. It presents the total number of server CPU seconds used in the benchmark as a function of load for four different configurations. The overhead due to replication is the difference between the curves labelled "Coda:3" and "Coda: NoRep". The overhead due to Camelot is the difference between the curves labelled "Coda: NoRep" and "Coda: NoCam". The latter curve corresponds to the configuration "Coda: NoRep" in which we substituted a dummy transactional virtual memory package for Camelot. The dummy had an interface identical to Camelot but incurred zero overhead on every Camelot operation other than transaction commit. For the latter operation, the dummy performed a *write* system call



This graph shows the total elapsed time for the benchmark as a function of load. Table 2 presents the same information in greater detail.

Figure 2: Effect of Load on Elapsed Time

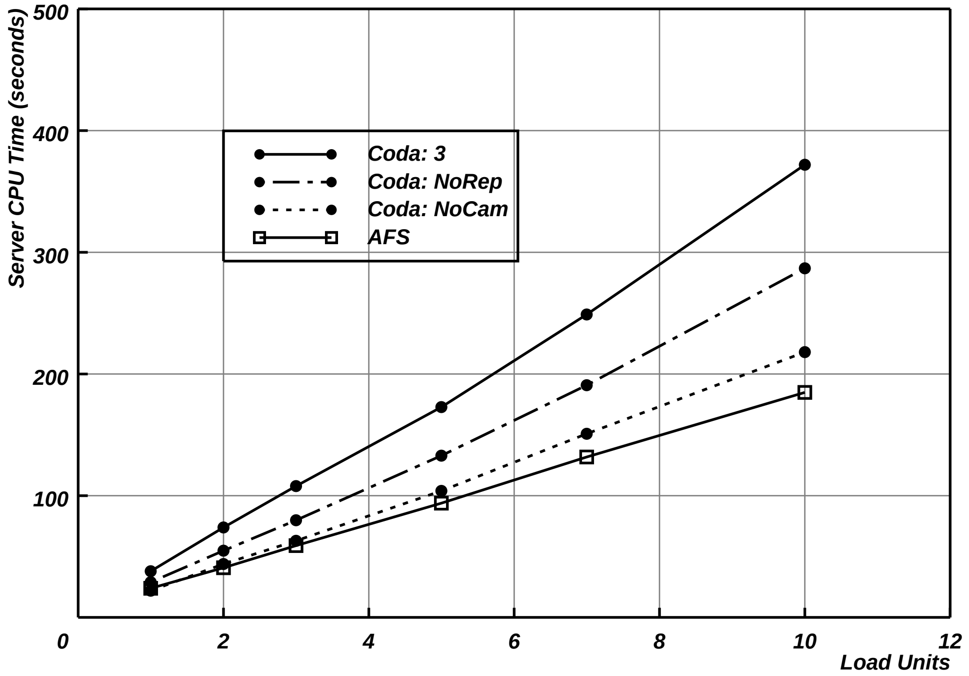
Configuration	Load Units	MakeDir	Copy	ScanDir	ReadAll	Make	Total
Coda: 3							
	1	5 (1)	48 (1)	33 (1)	52 (1)	248 (5)	386 (5)
	2	6 (1)	51 (1)	32 (0)	52 (1)	251 (6)	391 (4)
	3	8 (2)	56 (5)	32 (0)	51 (0)	267 (5)	414 (5)
	5	11 (2)	83 (7)	34 (0)	54 (0)	278 (8)	460 (7)
	7	15 (2)	114 (5)	33 (0)	53 (0)	313 (4)	529 (3)
	10	30 (1)	170 (3)	34 (0)	53 (0)	369 (9)	657 (8)
AFS							
	1	7 (3)	40 (1)	27 (2)	53 (0)	242 (4)	369 (7)
	2	6 (1)	41 (1)	27 (0)	54 (0)	243 (1)	369 (1)
	3	6 (1)	46 (2)	27 (1)	54 (0)	247 (1)	379 (2)
	5	6 (1)	44 (1)	27 (0)	53 (0)	251 (2)	382 (2)
	7	8 (1)	52 (1)	27 (1)	53 (1)	259 (0)	399 (1)
	10	10 (2)	65 (1)	27 (0)	52 (0)	275 (2)	429 (3)

This table compares the running time of the phases of the Andrew benchmark for Coda replicated at 3 servers with AFS. Each time reported is in seconds, and is the mean of 3 trials. Numbers in parentheses are standard deviations.

Table 2: Effect of Load on Elapsed Time

with an amount of data corresponding to the average amount of data logged during a Camelot transaction in Coda[8]. The curve thus indicates the expected performance of Coda for non-replicated data if a low-overhead transactional system were to be used in lieu of Camelot. The overhead due to lack of tuning in Coda is the difference between the curves labelled "Coda:NoCam" and "AFS".

Linear regression fits for the four configurations indicate slopes of 36.7, 28.5, 21.7 and 18.0 seconds per load unit respectively. In other words, each additional load unit increases server CPU utilization by these amounts in these configurations. The correlation coefficient is greater than 0.99 in each case, indicating that a linear model is indeed an excellent fit over this range of load.



This graph shows total number of server CPU seconds used as a function of load in running the benchmark. The configuration corresponding to the curve labelled "Coda:NoCam" is described in the text of Section 8.3.

Figure 3: Effect of Load on Server CPU Utilization

8.4. Effect of Multicast

Early in our design we debated the importance of multicast, perceiving both advantages and disadvantages in its use. To quantify the contribution due to multicast we repeated the load experiment with multicast turned off. Clients and servers communicated via the non-multicast version of MultiRPC for this experiment.

Multicast is beneficial in two ways. It reduces the latency of storing large files, and it reduces network load. Since the Andrew benchmark does not involve very large files, we did not observe a substantial improvement in latency due to multicast. But we did observe substantial reduction in network load. Figures 4 shows the total number of bytes transmitted as a function of load during the running of the benchmark. As one would expect for a replication factor of 3, multicast reduces the number of bytes transmitted by about two-thirds. Figure 5 shows the corresponding number of packets transmitted. The improvement due to multicast is less dramatic than in Figure 4 because many small non-multicast control packets are transmitted as part of the multicast file transfer protocol.

---

[8]Although a Unix *write* is only synchronous with the copying of data to a kernel buffer, the comparison is fair because the Camelot log was on a ramdisk for our experiments.

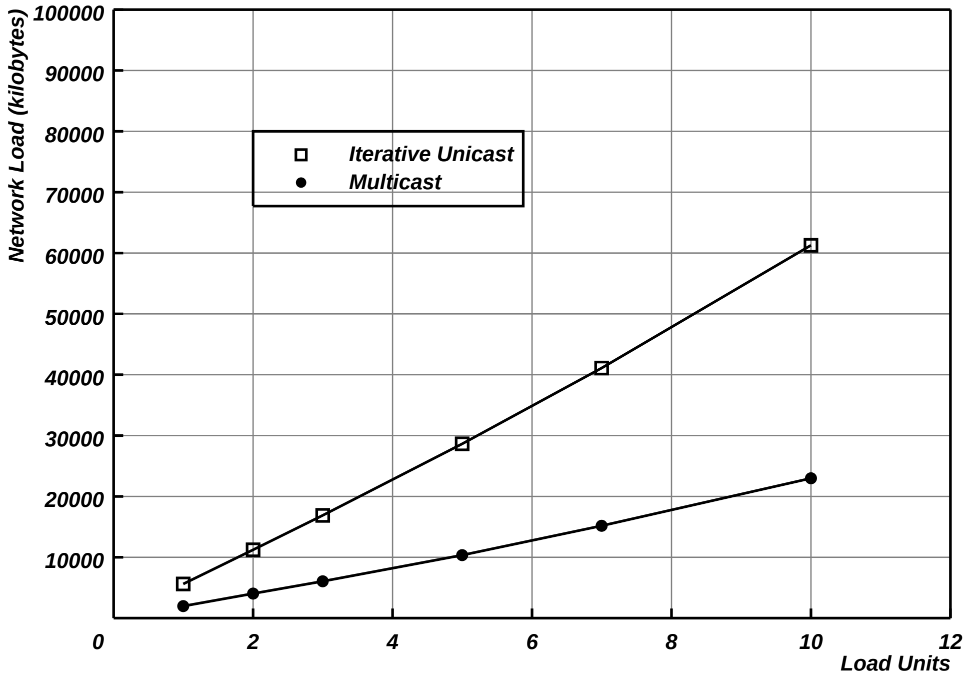


Figure 4: Effect of Multicast on Bytes Transmitted

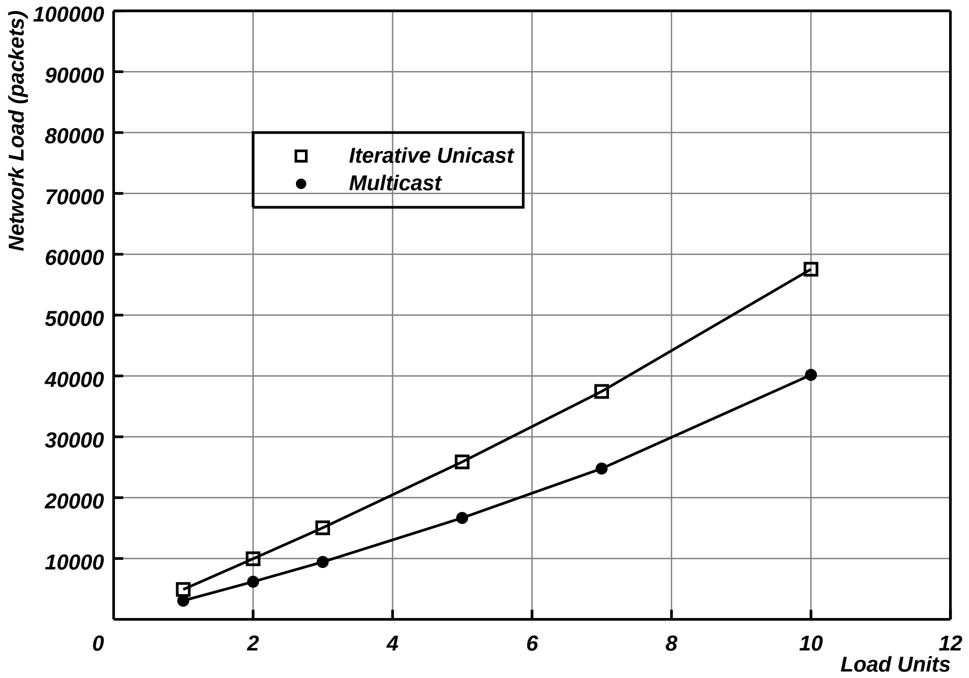


Figure 5: Effect of Multicast on Packets Transmitted

8.5. Contribution of Ramdisk

The function of the ramdisk is to reduce the cost of log forces in Camelot, thereby reducing the latency of update operations. Since the MakeDir and Copy phases of the benchmark involve the most updates, their combined running time is an indicator of the contribution of the ramdisk. We measured these times for two configurations: one in which Camelot uses a raw disk partition for a log, and the other where it uses a ramdisk. The data shows that it takes 58 seconds in the former case and 53 seconds in the latter, a reduction of about 9%.

9. Related Work

The system most closely related to Coda is undoubtedly its ancestor, AFS. Coda strives to preserve the virtues of AFS while significantly enhancing its availability. The specific design decisions inherited from AFS have been described in Section 2.1.

Data availability has been the topic of many research efforts in the last decade. A few of these have been experimental projects to provide high availability in distributed file systems. Examples of such projects include Violet [4, 5], RNFS [8] (based on ISIS [1]), Saguaro [12] and Locus [19, 11]. All of these, with one exception, have used a pessimistic approach to replica management and have therefore had little influence on Coda.

The exception is Locus, originally developed as a research prototype at UCLA and now marketed by Locus Computing Corporation. There are significant differences in the research and commercial versions of Locus. Most importantly, optimistic replication is only used in the research version of Locus. A less ambitious primary-site replication scheme is used in the commercial version. In the rest of this section the term ‘‘Locus’’ specifically refers to the research version.

Coda uses three ideas from Locus:

- The view that optimistic replication is acceptable in a Unix environment.
- The use of version vectors for detecting conflicts.
- The use of Unix directory semantics to partially automate resolution.

But there are major differences between the two systems, the most significant of which are the following:

- Scalability and security are fundamental goals in Coda, but not in Locus.
- Only Coda explicitly addresses the use of portable computers.
- Coda is based on the client-server model, while Locus assumes a peer model.
- Coda integrates the use of two different mechanisms, whole-file caching and replication, while Locus relies solely on replication.
- Coda clients directly update server replicas in parallel. A Locus usage site updates a single replication site which then notifies other replication sites. The latter sites asynchronously update themselves from the first replication site.
- Coda provides an approximation to Unix semantics. Locus provides an exact emulation of Unix semantics that is less scalable and considerably more complex to implement.

The differences between Coda and Locus can be traced to their fundamentally different views of distributed computing systems. Locus was designed in the early 1980’s, prior to the advent of workstations, when distributed systems typically consisted of a few sites. Coda, in contrast, is specifically designed for distributed workstation environments which are already one to two orders of magnitude larger in scale. This difference in scale is the origin of virtually every design difference between Coda and Locus.

Coda uses a highly dynamic mechanism, caching, to reduce network and server load, to support user mobility, and to enable the addition and removal of clients at will. The more static mechanism, replication, is only used among the far fewer number of servers. Clients only have to keep track of the accessibility of servers, not of other clients. Updating server replicas directly by clients reduces total server CPU load, the resource most widely reported to be the performance bottleneck in large-scale distributed file systems. In contrast, we do not see how the strategies of Locus can be easily adapted for use in a large-scale distributed environment.

A performance comparison between Coda and Locus would be valuable in understanding the merits of the two systems. But in spite of the voluminous literature on Locus, there is no quantitative assessment of the performance implications of its high availability mechanisms.

10. Conclusion

Our goal in building Coda is to develop a distributed file system that retains the positive characteristics of AFS while providing substantially better availability. In this paper we have shown how these goals have been achieved through the use of two complementary mechanisms, server replication and disconnected operation. We have also shown how disconnected operation can be used to support portable workstations.

Although Coda is far from maturity, our initial experience with it reflects favorably on its design. Performance measurements from the Coda prototype are promising, though they also reveal areas where further improvement is possible. We believe that a well-tuned version of Coda will indeed meet its goal of providing high availability without serious loss of performance, scalability or security. A general question about optimistic replication schemes that remains open is whether users will indeed be willing to tolerate occasional conflicts in return for higher availability. Only actual experience will provide the answer to this.

Acknowledgments

We thank Eric Cooper and Lily Mummert for their useful comments on this paper, Monique Benarrosh and Kimberly Ginther-Webster for finding many elusive Locus documents, the members of the Camelot group for helping us use their system, and the AFS group for providing us with the software base on which we have built Coda.

References

1. Birman, K.P. and Joseph, T.A. "Reliable Communication in the Presence of Failures". *ACM Transactions on Computer Systems 5*, 1 (February 1987).

2. Davidson, S.B. "Optimism and Consistency in Partitioned Distributed Database Systems". *ACM Transactions on Database Systems 9*, 3 (September 1984).

3. Davidson, S.B., Garcia-Molina, H., Skeen, D. "Consistency in Partitioned Networks". *ACM Computing Surveys 17*, 3 (September 1985).

4. Gifford, D.K. Violet, an Experimental Decentralized System. Tech. Rept. CSL-79-12, Xerox Corporation, Palo Alto Research Center, September, 1979.

5. Gifford, D.K. Weighted Voting for Replicated Data. Tech. Rept. CSL-79-14, Xerox Corporation, Palo Alto Research Center, September, 1979.

6. Guy, R.G.II. A Replicated Filesystem Design for a Distributed Unix System. Master Th., Department of Computer Science, University of California, Los Angeles,1987.

7. Howard, J.H., Kazar, M.L., Menees, S.G., Nichols, D.A., Satyanarayanan, M., Sidebotham, R.N., West, M.J. "Scale and Performance in a Distributed File System". *ACM Transactions on Computer Systems 6*, 1 (February 1988).

8. Marzullo, K., Schmuck, F. Supplying High Availability with a Standard Network File System. Proceedings of the 8th International Conference on Distributed Computing Systems, San Jose, June, 1988.

9. Morris, J. H., Satyanarayanan, M., Conner, M.H., Howard, J.H., Rosenthal, D.S., Smith, F.D. "Andrew: A Distributed Personal Computing Environment". *Communications of the ACM 29*, 3 (March 1986).

10. Parker, D.S. Jr., Popek, G.J., Rudisin, G., Stoughton, A., Walker, B.J., Walton, E., Chow, J.M., Edwards, D., Kiser, S., Kline, C. "Detection of Mutual Inconsistency in Distributed Systems". *IEEE Transactions on Software Engineering SE-9*, 3 (May 1983).

11. Popek, G.J., Walker, B.J.. *The LOCUS Distributed System Architecture.* MIT Press, 1985.

12. Purdin, T. *Enhancing File Availability in Distributed Systems (The Saguaro File System)*. Ph.D. Th., University of Arizona, 1987.

13. Satyanarayanan, M. "Integrating Security in a Large Distributed System". *ACM Transactions on Computer Systems 7*, 3 (August 1989).

14. Satyanarayanan, M., Siegel, E.H. "Parallel Communication in a Large Distributed Environment". *IEEE Transactions on Computers to appear* (1990).

15. Satyanarayanan, M., Howard, J.H., Nichols, D.N., Sidebotham, R.N., Spector, A.Z., West, M.J. The ITC Distributed File System: Principles and Design. Proceedings of the 10th ACM Symposium on Operating System Principles, Orcas Island, December, 1985.

16. Satyanarayanan, M. A Survey of Distributed File Systems. In *Annual Review of Computer Science*, Traub, J.F., Grosz, B., Lampson, B., Nilsson, N.J., Eds., Annual Reviews, Inc, 1989. Also available as Technical Report CMU-CS-89-116, Department of Computer Science, Carnegie Mellon University, February, 1989.

17. Spector, A.Z., Kazar, M.L. "Wide Area File Service and the AFS Experimental System". *Unix Review 7*, 3 (March 1989).

18. Spector, A.Z., Swedlow, K.R. ed. *The Guide to the Camelot Distributed Transaction Facility: Release 1.* 0.98(51) edition, Carnegie Mellon University, 1988.

19. Walker, B., Popek, G., English, R., Kline, C., Thiel, G. The LOCUS Distributed Operating System. Proceedings of the 9th ACM Symposium on Operating System Principles, Bretton Woods, October, 1983.

Table of Contents

List of Figures

List of Tables